

# References and Local State

Mattox Beckman  
beckman@iit.edu

Illinois Institute of Technology

## Definition

### §1 Equational Reasoning

The rule of *referential transparency*:

$$\frac{e_1 \Downarrow v \quad e_2 \Downarrow v \quad f \ e_1 \Downarrow w}{f \ e_2 \Downarrow w}$$

- If you have two expressions that evaluate to be the same thing then you can use one for the other without changing the meaning of the whole program.
- e.g.  $f(x) + f(x) == 2 * f(x)$
- You can prove this by induction, using the natural semantic rules from the previous lectures.

## Objectives

### §0 Objectives

The lack of mutable variables gives us the ability to perform many analyses using mathematics. In this lecture we talk about *equational reasoning* and references, and see techniques for limiting the scope of the state to improve the reliability of your code.

- Be able to explain equational reasoning and give an example.
- Know the syntax of references in OCaml.
- Know the tradeoffs between imperative and functional features.
- Know the constructions to define a function with local state.
- Be able to state the benefits of local state and give an example.
- Be able to use tuples to allow multiple functions access to the same state.

### §1 Equational Reasoning

- You can use equational reasoning to make the following equivalence:

$$f(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \equiv \text{if } e_1 \text{ then } f(e_2) \text{ else } f(e_3)$$

```
1 x * (if foo then 20 / x else 23 / x) equivalent to
2 if foo then 20 else 23    (well, mostly)
```

- You have the basis now of many compiler optimization opportunities!

```

1 # let counter = something
2 val counter : unit -> int = <fun>
3 # counter ();;
4 - : int = 1
5 # counter ();;
6 - : int = 2
7 # counter ();;
8 - : int = 3
9 #

```

- Can we still use equational reasoning to talk about programs now?

### Transition Semantics

$\text{ref } v \rightarrow \$i$ , where  $\$i$  is a free location in the state, initialized to  $v$ .

$! \$i \rightarrow v$ , if state location  $\$i$  contains  $v$

$\$i := v \rightarrow ()$ , and state location  $\$i$  is assigned  $v$ .

$() ; e \rightarrow e$

Note that references are different than pointers: once created, they cannot be moved, only assigned to and read from.

$f(x) + f(x) == 2 * f(x)$

```

1 # 2 * counter ();;
2 - : int = 8
3 # counter () + counter ();;
4 - : int = 11

```

- Congratulations. You just broke mathematics.

$\frac{e \Downarrow v}{\text{ref } e \Downarrow \$i}$ , where  $\$i$  is a free location in the state, initialized to  $v$ .

$\frac{e \Downarrow \$i}{!e \Downarrow v}$ , if state location  $\$i$  contains  $v$ .

$\frac{e_1 \Downarrow \$i \quad e_2 \Downarrow v}{e_1 := e_2 \Downarrow ()}$ , and location  $\$i$  is set to  $v$ .

$\frac{e_1 \Downarrow () \quad e_2 \Downarrow v}{e_1 ; e_2 \Downarrow v}$

```
1 # let ct = ref 0;;
2 val ct : int ref = {contents=0}
3 # let counter () =
4     ct := !ct + 1;
5     !ct;;
6 val counter : unit -> int = <fun>
7 # counter ();;
8 - : int = 1
9 # counter ();;
10 - : int = 2
```

State is bad because:

- it breaks our ability to use equational reasoning
- users can get to our global variables and change them without permission

State is good because:

- Certain constructs are almost impossible without state (e.g., Graphs)
- Our world is a stateful one

`ct` is globally defined. Two bad things could occur because of this.

1. What if you already had a global variable `ct` defined?
  - Correct solution: use modules.
2. The Stupid User<sup>TM</sup> might decide to change `ct` just for fun.
  - Now your counter won't work like it's supposed to...
  - Now you can't change the representation without getting tech support calls.
  - Remember the idea of *abstraction*.

Review of scope:

```
1 let x = 10;;
2
3 let foo y = match y with
4 | 0, b -> let c = b * b in
5           let d = c * c in
6           b * c * d
7 | a, b -> map (fun z -> z + a + x) [a;b]
```

- `x` exists from line 2–7.
- `y` exists from line 3–7.
- `b` exists from line 4–6.
- `c` exists from line 5–6.
- `d` exists on line 6 only.
- `a` and `b` exist on line 7 only.
- `z` exists on line 7, after the `fun z` until the `)`.

```
1 # let counter =
2   let ct = ref 0 in
3   fun () -> ct := !ct + 1; !ct;;
4 val counter : unit -> int = <fun>
5 # counter ();;
6 - : int = 1
7 # counter ();;
8 - : int = 2
```

- This protects `ct`, making it available only to `counter`.

```
1 # let (counter, reset) =
2   let ct = ref 0 in
3   (fun () -> ct := !ct + 1; !ct),
4   (fun nv -> ct := nv);;
5 val counter : unit -> int = <fun>
6 val reset : int -> unit = <fun>
7 # counter ();;
8 - : int = 1
9 # reset 5;; (* This trick brought to you by *)
10 - : unit = () (* higher order functions, tuples, *)
11 # counter ();; (* and the principle of orthogonality. *)
12 - : int = 6
```

```
1 # let mkRandom s =
2   fun () -> s := (!s * 9 + 5) mod 1024; !s;;
3 val mkRandom : int ref -> unit -> int = <fun>
4 # let rnd0 = mkRandom (ref 1);;
5 val rnd0 : unit -> int = <fun>
6 # rnd0 ();;
7 - : int = 14
8 # rnd0 ();;
9 - : int = 131
10 # rnd0 ();;
11 - : int = 160
```

- In this version we pass the reference into the function rather than creating our own.

```
1 # let enumerate lst (ctfun, rsfun) =
2   rsfun 0;
3   List.map (fun x -> (ctfun (), x)) lst;;
4 val enumerate : 'a list ->
5   (unit -> 'b) * (int -> 'c) -> ('b * 'a) list = <fun>
6 # enumerate ["hello"; "there"; "class"]
7   (counter, reset);;
8 - : (int * string) list = [1, "hello"; 2, "there";
9   3, "class"]
10 #
```

- We can give the counter to another function.
- This is not good. Why not?

**Activity****§5 Activity**

1. Supposing you wanted a counter that did *not* use references, how would you go about writing it?
2. The random number function generator does not have a way to reset the state. We would also like to be able to ask “what was the last random number generated” without changing the seed. Write a (group of) functions to do this.
3. Suppose we want a more generic way to represent counters—in fact, suppose you want *several* counters in your program. You could just repeat the code several times, but there are serious flaws to that approach. What are they? How might you go about fixing them?

**Answer to 2****§5 Activity**

```

1 # let mkRandom init =
2   let seed = ref init in
3   (( fun () -> seed := !seed * 4; !seed),
4     ( fun ns -> seed := ns),
5     ( fun () -> !seed) );;
6   val mkRandom : int -> (unit -> int) * (int ->
7 # let (rnd, reset, last) = mkRandom 4;;
8 val rnd : unit -> int = <fun>
9 val reset : int -> unit = <fun>
10 val last : unit -> int = <fun>

```